

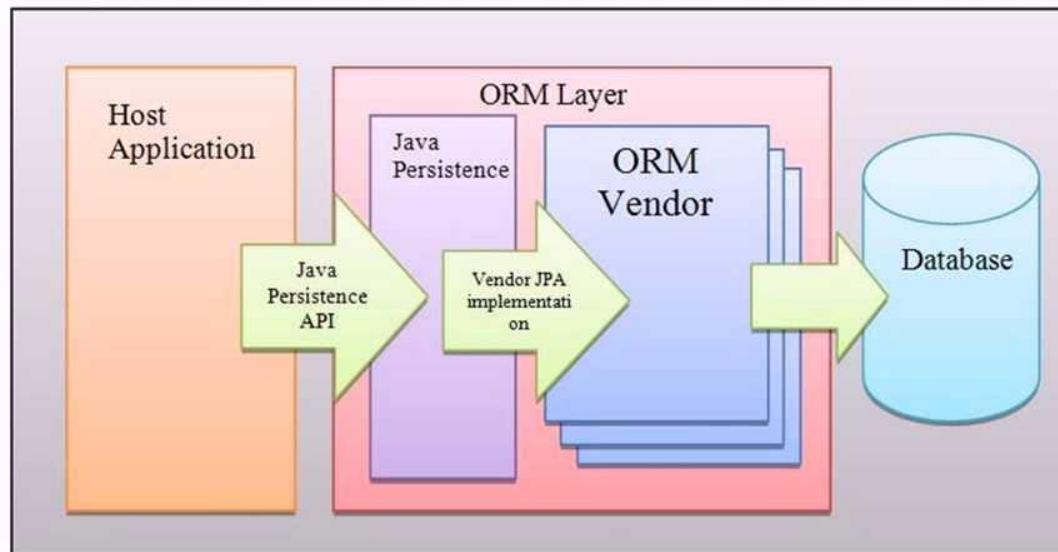
# Java EE

# JPA – Java Persistence API

Vortragende:  
Mag. Christian Schäfer  
Vertretung: DI Matthias Kapferer

# JPA Architecture

## Border view



# Persistenz

- ▶ Persistente Objekte
  - kapseln zusammengehörende Daten
  - werden von einer Datenbank geladen und gespeichert
- ▶ Persistenz-Frameworks
  - bieten einen Persistenz-Mechanismus für den Datenbank-Zugriff
  - O/R-Mapping
    - beschreibt die Zuordnung von Datenbank-Spalten zu Objekt-Eigenschaften

# JPA – Java Persistence API

- ▶ DAS Persistenz-Framework für alle Arten von Java Applikationen
  - Java Desktop Applikationen
  - Java Web Applikationen
  - Java Enterprise Applikationen

# JPA – Java Persistence API

- ▶ verwendet POJOs als persistente Objekte
  - "gewöhnliche" Java-Klassen werden mit Annotationen persistent gemacht
- ▶ für Persistenz in relationalen Datenbanken
- ▶ definiert eigene Abfragesprache
  - JPQL (Java Persistence Query Language)
  - SQL-ähnliche Query-Language
- ▶ definiert Schnittstellen für Persistenz-Provider (=die Datenbanksysteme)

# JPA – Implementierungen

- ▶ TopLink Essentials
  - ehem. Referenz-Implementierung von SUN
- ▶ EclipseLink
  - neue Referenz-Implementierung
  - in Glassfish enthalten
  - als Standalone Anwendung möglich
- ▶ Hibernate
  - weit verbreitete Third-Party-Implementierung
- ▶ ...

# Entities

- ▶ Basiseinheit der Persistenz
  - reguläre Java Klassen,
  - erweitert um
    - Identität (Primärschlüssel)
    - Mapping-Informationen von Datenbank-Spalten

# Entities – mögliche Datentypen

- ▶ Primitive Datentypen und ihre Wrapper, z.B.
  - int/Integer, long/Long, ...
- ▶ Serialisierbare Standard-Klassen wie
  - String, BigInteger, BigDecimal, Date, Time, Timestamp, Calendar
- ▶ Arrays z.B.
  - byte[], Byte[], char[], Character[]
- ▶ Aufzählungstypen (enums)
  - `enum MyEnum {value1, value2}`



# Entities – mögliche Datentypen

- ▶ Serialisierbare Klassen
  - `class AClass implements Serializable {...}`
- ▶ Entities für persistente Beziehungen (Fremdschlüssel)
  - Referenz auf Entity
  - Collection von Entities

# Entities definieren

Annotation	Bedeutung
@Entity	kennzeichnet eine Klasse als Entität nur für Toplevel Klassen mit Default-Konstruktor
@Table	spezifiziert den Tabellenamen für diese Entität
@SecondaryTable	zusätzliche Tabelle für die Datenherkunft der Entität
@PrimaryKeyJoinColumn	definiert die PK-Spalte der SecondaryTable
@Embeddable	Klasse kann Typ einer Property/eines Feldes in einer anderen Entity sein

# Entities definieren

Annotation	Bedeutung
@Column	spezifiziert Spaltennamen und Herkunftstabelle
@Id	kennzeichnet ein Feld/eine Property als Primärschlüssel
@EmbeddedId	kennzeichnet ein Feld/eine Property als Primärschlüssel (für zusammengesetzten Key)
@Basic	definiert Hinweise wie optional od. verzögertes Laden
@Enumerated	gibt an, wie Werte von Enum-Typen abgebildet werden (STRING oder ORDINAL)
@Temporal	gibt Zeit-Datentyp an (DATE, TIME, TIMESTAMP)
@Lob	für BLOBs und CLOBs (Large Objects)
@Transient	das Feld/die Property wird nicht persistiert

# Entities definieren

**@Entity**

```
public class Namen implements Serializable {  
    private int nr;
```

**@Id**

```
    public int getNr(){  
        return nr;  
    }  
    // weitere Felder werden automatisch gemappt
```

```
    private String name;
```

**@Basic**(fetch=FetchType.LAZY)

```
    private String adresse;
```

**@Transient**

```
    transient private boolean changed;
```

```
    ...
```

```
}
```

# Entities definieren

- ▶ Generieren von Primärschlüssel–Werten
  - **@GeneratedValue** – automatische Generierung von Primärschlüssel–Werten
    - AUTO/IDENTITY: der Provider wird den neuen Wert automatisch festlegen
    - SEQUENCE: ein Sequenzgenerator wird den Wert festlegen, dieser muss als **@SequenceGenerator** definiert werden
    - TABLE: eine Tabelle enthält den Wert, der als nächstes verwendet werden soll, diese muss als **@TableGenerator** definiert werden

# Entities definieren

**@Entity**

```
public class Gattungen implements Serializable {
```

**@Id**

```
@GeneratedValue(generator="TIERE_SEQ",  
strategy=GenerationType.SEQUENCE)
```

**@SequenceGenerator**(

```
    name="TIERE_SEQ", sequenceName="TIERE_SEQ", allocationSize=1)
```

```
private BigDecimal tiergattungsnr;
```

```
...
```

```
}
```

# Entities: Datum und Uhrzeit

SQL Typ	Java Typ
DATE	java.util.Date, java.sql.Date
TIME	java.util.Date, java.sql.Time
TIMESTAMP	java.util.Date, java.sql.Timestamp

SQL Typ kann mit  
**@Temporal(TemporalType)**  
überschrieben werden

SQL Typ kann mit  
**@Column(columnDefinition)**  
überschrieben werden

SQL Typ	Java Time API Type
DATE	LocalDate
TIME	LocalTime
TIMESTAMP	LocalDateTime
TIME WITH TIMEZONE	OffsetTime
TIMESTAMP W. TIMEZONE	OffsetDateTime

# Entities: Assoziationen

Annotation	Bedeutung
@OneToOne	definiert eine 1:1 Beziehung zu einer 2. Tabelle
@OneToMany	definiert die 1-Seite einer 1:n Beziehung (Collection mit Fremd-Schlüssel-Entities)
@ManyToOne	definiert die n-Seite einer 1:n Beziehung (Referenz auf Primärschlüssel-Entity)
@ManyToMany	definiert eine Seite einer n:m Beziehung
@JoinColumn	zur Angabe von Informationen zu einer Join-Spalte



# Entities: Assoziationen

```
@Entity public class Gattung implements Serializable {  
    @Id  
    private int gattungsId;  
    @OneToMany(mappedBy = "gattung")  
    private List<Tier> tiere;  
    ...  
}
```

```
@Entity public class Tier implements Serializable {  
    @Id  
    private int tierId;  
    @JoinColumn(name = "GattungsId")  
    @ManyToOne  
    private Gattungen gattung;  
    ...  
}
```

# Entities definieren – Alternative

```
<entity-mapping>
  <entity class="data.Namen" access="PROPERTY">
    <sequence-generator name="TIERE_SEQ"
                        sequence-name="TIERE_SEQ" />
    <id name="nr">
      <generated-value strategy="SEQUENCE"
                      generator="TIERE_SEQ" />
    </id>
  </entity>
</entity-mapping>
```

# Persistence Unit

- ▶ Persistenz-Archiv
  - fasst mehrere Entities zusammen
    - die zusammen gehören und
    - aus 1 Datenquelle stammen

# Persistence Unit

- ▶ wird mit Deployment Descriptor konfiguriert
  - Provider
    - welches Persistenzframework verwendet wird
  - Entities
    - class (kann mehrfach vorkommen)
    - jar-file: falls angegeben, werden die annotierten Klassen aus diesem File verwendet
  - Properties
    - Provider-Spezifische Informationen wie
      - Connection-Url, Username etc.
  - Dateiname:
    - META-INF/persistence.xml

# Persistence Unit

RESOURCE\_LOCAL = lokales  
Transaktionsmodell  
(JTA = Transaktionsmodell des  
Containers)

```
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="Zoo" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>model.Tier</class>
    <properties> <!-- Provider-Spezifische Informationen -->
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/Zoo" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="" />
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />
    </properties>
  </persistence-unit>
</persistence>
```

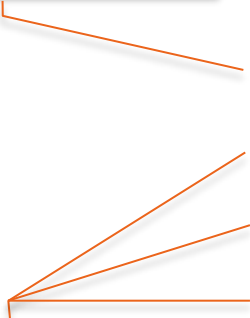
# EntityManager

- ▶ Schnittstelle zwischen Entity und Datenbank
  - Erzeugung durch EntityManagerFactory
  - enthält Methoden um Daten abzufragen und zu ändern

Methode	Beschreibung
createQuery	Daten abfragen
find	Datensatz nach Primary Key suchen
persist	ein Objekt hinzufügen
merge	Änderungen für ein existierendes Objekt übernehmen
remove	ein Objekt löschen

# EntityManager

- ▶ Transaktionsunterstützung
  - nur Änderungen, die in einer Transaktion ablaufen, werden zur Datenquelle übertragen
  - Bei lokaler Verwendung (transaction-type *RESOURCE\_LOCAL*) begin/commit/rollback selbst ausführen

EntityManager	Methode	Beschreibung
	getTransaction	liefert das Transaktionsobjekt
	begin	startet eine Transaktion
	commit	führt ein Commit aus
	rollback	führt ein Rollback aus
EntityTransaction		

# Transaktionen in Java EE

- ▶ Application-Managed Entity Manager
  - transaction-type="RESOURCE\_LOCAL"
  - Anwendung erzeugt den EntityManager selber
  - Transaktionsmanagement wie in lokaler Anwendung

```
@PersistenceUnit EntityManagerFactory emf;  
EntityManager em = emf.createEntityManager();
```



# Transaktionen in Java EE

- ▶ Container-Managed Entity Manager
  - `transaction-type="JTA"` mit `@TransactionManagement (CONTAINER)`
  - Transaktionen werden vom Container gemanagt
  - `@TransactionAttribute(TransactionAttributeType)` steuert das Verhalten der EJB/des Containers
    - `SUPPORTS`, `NOT_SUPPORTED`, `REQUIRED`, `REQUIRES_NEW`, ...
  - Entity Manager muss über Injection geholt werden  
`@PersistenceContext EntityManager em;`
  - dieselbe EntityManager-Instanz steht in allen beteiligten JavaEE Komponenten zur Verfügung

# Transaktionen in Java EE

- ▶ Container-Managed Entity Manager
  - transaction-type="JTA" mit  
@TransactionManagement (BEAN)
  - Transaktionen werden von der Bean gemanagt
  - Entity Manager muss über Injection geholt werden  
@PersistenceContext EntityManager em;
  - dieselbe EntityManager-Instanz steht in allen  
beteiligten JavaEE Komponenten zur Verfügung

# Transaktionen in Java EE

- ▶ Container-Managed Transactions in CDI
  - `@Transactional` Annotation
    - Erweitert Transaktions-Management auf beliebige CDI Beans
  - Entity Manager muss über Injection geholt werden  
`@Inject EntityManager em;`
  - Gleiche Transaktions-Typen wie bei EJBs:
    - `SUPPORTS`, `NOT_SUPPORTED`, `REQUIRED`, `REQUIRES_NEW`, ...

# JPQL

- ▶ Einheitliche Abfragesprache für alle DB Provider
  - Ähnlich SQL, aber auf Entities gerichtet
  - Ist case sensitive

```
// Alle Tiere
```

```
Select t from Tier t
```

```
// Alle Pflanzenfresser
```

```
Select t from Tier t where t.pflanzenfresser = true
```

```
// Alle Tiere mit einem bestimmten wert im Feld pflanzenfresser
```

```
Select t from Tier t where t.pflanzenfresser = :flag
```

# JPQL

```
// Alle Tiere mit m im Namen
```

```
Select t from Tier t where t.name like '%m%'
```

```
// Alle Tiere der Gattung Reptilien
```

```
Select t from Tier t JOIN t.gattung g where g.name='Reptilien'
```

```
// Alle Tiere mit ihrer Gattung
```

```
Select t from Tier t JOIN FETCH t.gattung
```

```
// Alle Tiere deren Gewicht NULL ist
```

```
Select t from Tier t where t.gewicht is null
```

```
// Alle Gattungen ohne Tiere
```

```
Select g from Gattung g where g.tiere is empty
```

```
// Tiere Gruppiert
```

```
Select t.pflanzenfresser, count(t.tierId) from Tier t  
group by t.pflanzenfresser
```

# Criteria API

- ▶ Java API für die Generierung von Queries
  - CriteriaBuilder
    - Wird vom EntityManager erzeugt
    - erstellt Expressions für verschiedene Operatoren (like, equal, gt, lt, ...)
  - CriteriaQuery<E>:
    - Wird vom CriteriaBuilder erzeugt
    - nimmt Expressions für Ergebnistyp, Filter, Sortierung entgegen
  - Typisierte Abfragen sind möglich
    - Dafür werden zu allen Entities Metamodel-Klassen generiert

# Criteria API

```
// get a CriteriaBuilder object
CriteriaBuilder builder = em.getCriteriaBuilder();

// create query object
CriteriaQuery<Tier> query = builder.createQuery(Tier.class);

// set the query root to the Tier entity
Root<Tier> root = query.from(Tier.class);

// specify type, filter and sort order
query.select(root);
query.where(builder.like(root.get("name"), "%z%"));
query.orderBy(builder.asc(root.get("name")));

// create the query
TypedQuery<Tier> q = em.createQuery(query);
```

# Bean Validation

- ▶ Zur Validierung von
  - Objekten und ihren Member
  - Methoden
  - Konstruktoren
- ▶ Funktioniert mit Annotationen

@AssertFalse / @AssertTrue

@Null / @NotNull

@Min / @Max

@DecimalMin / @DecimalMax / @Digits

@Size

@Future / @Past

@Pattern



# Bean Validation

```
public class Tier {  
    ...  
    @Size(min = 1, max = 50)  
    @NotNull  
    public String getName() {  
        return this.name;  
    }  
    @Min(0)  
    @Max(10000)  
    public Integer getGewicht() {  
        return this.gewicht;  
    }  
    @Past()  
    public Date getImZooSeit() {  
        return this.imZooSeit;  
    }  
    ...  
}
```

```
public class Tier {  
    ...  
    @Size(min = 1, max = 50)  
    @NotNull  
    private String name;  
  
    @Min(0)  
    @Max(10000)  
    private Integer gewicht;  
  
    @Past()  
    private Date imZooSeit;  
    }  
    ...  
}
```

Annotationen können bei Property oder Feld gesetzt werden.

# Bean Validation

- ▶ Validierung erfolgt
  - Explizit durch Validator
    - liefert Liste von ConstraintViolation-Objekten
  - Implizit durch Framework, z.B.
    - JPA beim Speichern
      - Löst Exception aus
    - JSF bei der Validierung des User-Input
      - Fehler werden in der Eingabemaske angezeigt

# Bean Validation

```
// create and fill an entity
Tier entity = ...;

// get the validator factory
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
// get a validator
Validator validator = factory.getValidator();
// validate the entity
Set<ConstraintViolation<T>> violations = validator.validate(entity);
// process any validation errors
if (violations.size() == 0) {
    System.out.println("No validation problems found");
} else {
    violations.forEach(System.out::println);
}
```

# Method/Constructor Validation

- ▶ Validation Annotations können auch verwendet werden für
  - Argumente von Konstruktoren
  - Argumente von Methoden
  - Returnwert von Methoden
- ▶ Impliziter Aufruf durch Frameworks
  - Z.B. CDI (Contexts and Dependency Injection)
- ▶ Expliziter Aufruf
  - mit ExecutableValidator möglich
  - aber nicht empfohlen